Blockchain as a Trusted Component in Cloud SLA Verification

Amir Teshome Wonjiga
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Sean Peisert
CRD, LBNL
Berkeley, CA, USA

Louis Rilling
DGA
Rennes, France

Christine Morin
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

ABSTRACT

Migrating an application from local compute resources to commercial cloud resources involves giving up full control of the physical infrastructure, as the cloud service provider (CSP) is responsible for managing the physical infrastructure, including its security. The reliance of a tenant on a CSP can create a trust issue around whether the CSP is upholding its end of the bargain. CSPs acknowledge this and provide a guarantee through a Service Level Agreement (SLA). SLAs need to be verified for satisfaction of the defined objectives. To avoid raising the trust issue again, such a verification procedure needs to be unbiased and independently achievable by both tenants and CSPs without one relying on the other party.

In this paper, we consider an SLA offered by the provider that guarantees the integrity of tenants' data, and propose to verify the SLA using an integrity checking method based on a distributed ledger. Our proposed method allows both CSPs and tenants to perform integrity checking without one party relying on the other. The method uses a *blockchain* as a distributed ledger to store evidence of data integrity. Assuming the ledger as a secure, trusted source of information, the evidence can be used to resolve conflicts between providers and tenants. In addition, we present a prototype implementation and an experimental evaluation to show the feasibility of our verification method and to measure the time overhead.

KEYWORDS

Cloud, SLA, data integrity, remote integrity checking, blockchain

ACM Reference Format:
Amir Teshome Wonjiga, Sean Peisert, Louis Rilling, and Christine Morin. 2019. Blockchain as a Trusted Component in Cloud SLA Verification. In *IEEE/ACM 12th International Conference on Utility and Cloud Computing Companion (UCC '19 Companion), December 2–5, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3368235.3368872

1 INTRODUCTION

In the cloud, depending on the service model, the tenant and the cloud service provider (CSP) own different parts of the system. For instance, in a Storage as a Service cloud the provider owns the physical infrastructure and the virtualization layer while the tenant owns the data stored in the infrastructure. When migrating to the cloud the tenant loses full control of the physical infrastructure. The CSP is responsible for managing the infrastructure and so it monitors every aspect from resource allocation and performance to security. This creates a trust issue between CSPs and tenants.

CSPs acknowledge the trust issue and endeavor to provide assurance through an agreement called *Service Level Agreement (SLA)*. SLAs help in building a trustworthy relationship between CSPs and tenants by providing a guarantee for tenants as reward (i.e penalty) in case of SLA violation. An SLA is negotiated, hence users can specify their requirements to get a service well tailored to their needs. The SLA document describes the provided service, the rights and obligations of all participants and a quantitative description of the targeted quality of service (Service Level Objective, SLO).

SLOs must be verified for their fulfillment with any violation resulting in a penalty for the violating party. Ideally, after signing the agreement any participant should have the option to perform SLO verification independently of the other party(ies). Moreover, in case of violation, one party should be able to prove the violation to the others. Currently, detecting SLO violations is left as a responsibility for tenants. Even when tenants discover any violation, penalties do not apply automatically. Service providers perform checks on their side and the penalty is applied only if the provider discovers the violation. For example, the Amazon availability SLA [2] describes the procedure to report a violation and states *"if the Monthly Uptime Percentage of such request is confirmed by us and is less than the Service Commitment, then we will issue the Service Credit to you ..."*.

In our previous studies [26, 27], we described a mechanism to define and verify security monitoring terms in cloud SLAs. Specifically, we defined an SLA which guarantees the performance of *network intrusion detection systems* for a given set of vulnerabilities. The vulnerabilities are chosen based on tenants needs and requirements. Our verification process [26] requires cooperation between tenants and providers in order to verify the satisfaction of an SLO. In the context of SLAs, such dependency creates a conflict of interest and it requires trust from the other party, which goes back to the initial trust issue. Thus, we need a mechanism to reduce (remove if possible) such dependency in SLA verification.

To remove such dependency we need either a trusted third party or an independent system in which participants in the SLA do not have to trust one another for SLA verification. Recently there is an increase in application of distributed system technologies to make a secure system state change between untrusted parties without using any third party. These technologies, referred to as *blockchains*, are widely adopted for their property of *tamper-proof evidence* [23]. They are used as the core technology in digital currency (cryptocurrency) applications. They are also being used in different applications, e.g Internet of Things, health, identity etc.

In this paper, we provide an SLA verification approach for clouds which relies on the blockchain technology. In this process, tenants can perform verification at any given time. Besides, tenants

UCC '19 Companion, December 2–5, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-7044-8/19/12…$15.00
https://doi.org/10.1145/3368235.3368872

directly participate in keeping and maintaining attestation data secure, the data which is used for verification. Tenants participation is made possible by the ability of blockchains to be geographically distributed. Tenants can be part of the blockchain's network and participate in the SLA life-cycle process. We present an SLA verification method where tenants and providers participate in a process to keep verification data secure, which allows trusting the system in general rather than a single entity. Depending on the algorithm used in the blockchain operation process, some amount of participants can be malicious and the system can still be trusted.

We define SLAs guaranteeing *integrity* property of data stored in the cloud. Our goal is to show that having a trusted component in the SLA life-cycle helps to facilitate and improve the trust level of different phases in the life-cycle of SLA. In this paper we use data integrity property as an example, the method can also be used for other types of SLAs (e.g security monitoring SLAs [27]).

The rest of the paper is organized as follows. Section 2 describes the life-cycle of an SLA guaranteeing data integrity. The problem description is presented in Section 3. Section 4 presents background and related works. Section 5 presents the threat model for the verification process. The integrity checking process is presented in Section 6. Implementation and evaluation details are presented in Section 7 and 8 respectively. Finally, Section 9 concludes the paper.

2 SLA LIFE-CYCLE FOR MONITORING DATA INTEGRITY

The SLA life-cycle can be divided into three phases namely *SLA definition and negotiation, enforcement*, and *verification*. The contents of each phase of the SLA life-cycle is different depending on the properties covered by the SLA. *SLA definition and negotiation* is the phase where the two parties negotiate and define a set of terms for the provided service. In the *enforcement* phase providers implement the required mechanisms to enforce terms that are defined in the previous phase. The *verification* phase allows both providers and tenants to check whether the terms are correctly enforced or not.

We consider SLAs addressing the integrity property of data. Currently, most SLAs only address availability aspects (e.g., Amazon S3 [3] and DriveHQ [7]). For example, Amazon S3 claims to have "extremely durable" storage with redundancy and checks for corruption while data is at rest and in the network. However, Amazon SLA does not guarantee this property.

One of the major challenges to include security terms in SLAs is the difficulty in quantitatively measuring security properties like integrity and confidentiality. In our use case, tenants require checking either their data is corrupted or not. Hence, we assume the definition of SLAs with an objective to keep the data uncorrupted for the SLA lifetime. Other properties like backup frequency and type of access control policies can be included in the SLA definition.

Data corruption can occur due to various reasons. Depending on the cause of a corruption, there exist different detection and correction mechanisms [18–20].

Monitoring SLAs guaranteeing data integrity means continuously verifying the correctness of data stored in the cloud. Either the tenant or the provider can perform the verification. In this paper, we want to show a mechanism to perform verification without relying on the provider. We use a blockchain to store evidences which validate the correctness of outsourced data.

3 PROBLEM DESCRIPTION

Currently existing SLAs lack security terms, except for availability. The lack of security support has been a significant difficulty for the adoption of cloud services mainly for enterprises and cautious consumers (e.g Dropbox [8] uses Amazon's S3 service to store their clients data). Such a relationship between companies requires an agreement which covers the security aspect of the data.

Data integrity failure is a common issue in data storage systems [9, 14, 25]. There are different solutions to protect and recover from data corruption. In addition, in some fields of research and development, it is mandatory to keep and verify data integrity for others. For example, regulatory agencies controlling medical drugs and health care products publish data integrity guidelines [11]. Such agencies require implementing these guidelines in testing, manufacturing, packaging, distribution, and monitoring of drugs to review the quality and related issues of the product.

There are different reports of data integrity failure in production environments. In 2009 Facebook temporarily lost more than 10% of photos in hard drive failures [9]. Amazon S3 suffered from a data corruption issue [14] caused by a load balancer, which resulted mismatch of MD5 hash values supplied by the user. Wang et.al [25] examined 138 data corruption incidents reported in the bug repositories of four Hadoop projects. The study presented conclusions on the causes and impacts of data corruption and listed limitations in detection and handling of data corruption mechanisms. Moreover, data corruption may lead up to service unavailability. As presented by the technology website MacObserver [5] corrupted Apple iCloud data was a cause for an Apple iOS home screen crashing bug.

Guaranteeing data integrity through SLAs requires a verification mechanism that tenants can trust. For current commercial cloud SLAs, monitoring is performed by tenants or third-party companies, and service providers should confirm the violation. In order to minimize the trust issue between service providers and tenants, we need an open (non-secretive) process to do verification and to store and share the result without any bias. In this paper, we show a monitoring mechanism that can be used to check the correctness of data stored in the cloud without relying on the service provider.

Having different owners in different tiers or layers of the cloud is a challenge for verification. Verification in such an environment means checking the status of a service from a tier which does not belong to the verifier. In our case, the provider infrastructure holds the data and a tenant wants to check the integrity of that data.

We consider an SLO to keep the data uncorrupted throughout the validity period of the contract. Our problem is similar to a well-known problem called *remote data integrity checking*, which enables a server to prove to an auditor the integrity of a stored file. However, in our case the verification is in two ways, i.e. tenants need to check data integrity in the cloud and providers need to check the correctness of SLO violation claims submitted by tenants.

4 BACKGROUND AND RELATED WORK

We propose a verification mechanism based on the blockchain technology. This section presents an introduction to the blockchain technology and related works on data integrity monitoring tools.

4.1 Blockchain

Blockchain is a distributed linked list data structure, where each block contains a cryptographic hash of its predecessor, hence forming a chain. The chain is stored in a *distributed manner* and each valid block in the list is available in all participating node. Adding a new block requires an agreement between the participants. Participants use a consensus algorithm to decide what to add next in the chain. The distribution and the consensus algorithm make the system trustworthy without trusting any specific participant.

Blocks in a blockchain contain at least two parts. (i) *Link to a previous block:* the hash value of the previous block will be included in the current block, and it serves as a link from the previous to the current block. (ii) *Data stored in the current block:* is the information stored in the block. The data can be anything depending on the application using the blockchain. For example, in Bitcoin [23] the data section stores transactions.

Based on write permissions (permission to add blocks) blockchains can be grouped into three categories [12]. (i) *Public blockchain:* where anyone can join the network and participate in the consensus process to add blocks. (ii) *Private blockchain* where only one entity (e.g. the administrator) is allowed to add blocks. (iii) *Consortium blockchain* where only a pre-defined set of nodes are allowed to add blocks. These could also be generalized into two groups, *permissioned and permissionless.* Consortium and Private blockchains are permissioned while Public blockchains are permissionless.

After the first well known application of blockchain (Bitcoin [23]), there have been different implementations. We used a permissioned blockchains implementation called Hyperledger Fabric [16].

4.2 Related Works

This section presents related works on remote data integrity checking and the usage of blockchains for data integrity.

4.2.1 Remote data integrity checking. Integrity checking can be performed by downloading the whole data and compare it with a local version but this method contradicts the basic idea of the cloud, and it is impractical. Thus, almost all remote integrity checking works rely on cryptographic methods without any downloading.

There are two main approaches for remote data integrity checking: integrity checking *with and without a third party auditor (TPA).* TPA is a trusted party who has expertise for performing integrity checks and convincing both the client and the provider. TPA performs activities like generating a hash value for data blocks and comparing signatures to verify the integrity of stored data.

For example, TPA is used in a remote data possession checking protocol proposed by Luo and Bai [22]. They defined four functions namely *KeyGen, SigGen, GenProof, VerifyProof*: *KeyGen* run by users to generate a key, *SigGen* used by users to generate verification metadata, *GenProof* run by service providers to generate a proof of data storage correctness and *VerifyProof* used by TPA to verify the proof from service providers. The protocol executes in *setup* and *audit* phases. In the setup phase, users generate a key using the *KeyGen* and metadata for data to be sent to the cloud. The user can delete the local copies of the uploaded data after sending the data to the cloud and publishes the metadata to TPA.

The audit phase starts when the TPA commences the verification process. TPA formulates and sends a challenge to the provider and waits for a response. Upon receiving the challenge, the provider runs *GenProof* to generate a proof showing the data is correctly stored and sends back the proof to TPA. The TPA runs *VerifyProof* using the returned value and checks the result with the original metadata. This way TPA is used to check data integrity in the cloud.

Other published papers by Apolinario et al. [17], Zhu et al. [28] and Hao et al. [19] follow a similar pattern, but they use different cryptographic methods to achieve properties other than integrity, e.g data dynamics, privacy against verifiers, and proof on multiple providers. The protocol by Zhu et.al [28] supports data dynamics, and the work presented by Hao et.al [19] supports privacy against third-party verifiers. In some works the TPA is optional and the user can also be an auditor. However, in other works, the TPA is required (e.g Apolinario et al. [17]) and acts as a trusted third party (TTP). It is used to resolve disputes between the provider and tenants.

There are some works, for example by Hao et.al [20], that do not rely on a TPA. Hao et.al [20] follows a similar procedure as described above using six functions to operate, except the cryptographic procedures are different. Such protocols cannot be used for SLA verification because the process is only one way, i.e. only one party is performing the integrity check. For our use case both parties need to perform the verification. CloudProof [24] presents a storage system which provides proofs of a violation; hence neither providers nor tenants can bring a false claim of violation. Our work can be used to extend such a system in order to exploit the distributed nature of blockchains and become even more independent from other entities, like certificate authorities.

In our work, we want to reduce dependency between participants and possibly remove TTPs as it requires as much trust as providers. We thus propose to rely on a secure and distributed ledger.

4.2.2 Blockchains on data integrity. There are examples of data integrity services in IoT, database systems and data provenance tools. Recently, blockchain based storage systems (e.g Storj [1]), which feature data integrity by design are being developed. Liu et.al [21] proposed a blockchain-based data integrity framework for IoT data stored in the semi-trusted cloud. The framework incorporates data generators and data consumers and enables consumers to perform integrity verification. The framework uses a blockchain to store hash values while the the actual data is stored in a cloud.

Gaetani et.al [18] presented a blockchain-based database with strong integrity guarantees. They used two layers of blockchains, the first layer with a lightweight distributed consensus protocol that assures low latency and high throughput. The second layer is designed with a strong consensus to guarantee better integrity by using a proof of work (PWA) based algorithm. There are new upcoming companies (e.g Chainpoint [4]) offering to anchor users data to existing blockchains, which helps to verify the integrity and existence of data without relying on a trusted third party.

These studies do not describe the ability to verify integrity by other parties (other than the data owner). This is mainly a result of the considered threat model. For our use case, both parties require performing verification. In that sense, we are addressing a different kind of problem than most blockchain-based applications.

In our work, we want to show the use of a trusted component, like a blockchain, in the SLA verification process in order to reduce the dependency between tenants and providers and enable independent verification by both parties.

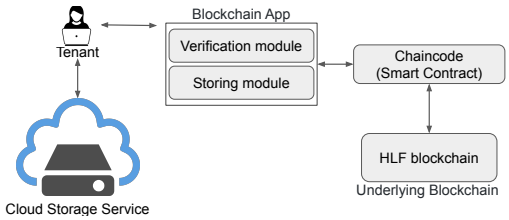


Figure 1: Data integrity verification architecture

5 THREAT MODEL

We assume that providers are semi-trusted. In a sense, a provider may return wrong data or wrong results of computation for a request from a tenant because an attacker altered the data or because of some other error. However, providers are not actively trying to alter tenants' data. In addition, providers never lie when claiming that they have not accepted to store some data. They are working towards maximizing profit and errors on the stored data are unintentional. Providers also have an economic incentive related to an SLA. Hence, they may lie in order not to violate an SLO.

Tenants store data in the cloud and do not keep a copy of the data locally. Tenants store a hash of the complete data on the blockchain to be used later for verification. Tenants may falsely claim a reward for a data integrity breach. Hence, providers need to do verification by themselves. However, tenants never lie when claiming that providers accepted to store data. A tenant interacts with a single provider, and the provider replies to requests about the data.

We assume at least the minimum number of required participants in the blockchain network are honest and are not controlled by the provider. In addition, we assume there is a secure communication network between tenants, providers, and the ledger. Notably, the integrity of messages is respected, i.e. there is no man-in-the-middle which is actively altering the communication between parties.

6 DATA INTEGRITY CHECKING

This section describes the proposed method to check integrity using a trusted, secure ledger and without relying on a third party. This method is used to perform SLA verification. Using this method, a tenant can check the integrity of the data stored in the cloud, and a provider can check the correctness of SLO violation claims without relying on the other party. Our method guarantees to link the signees with the same piece of evidence for data integrity.

We use a blockchain to remove the trusted third party. This secure ledger is used to store a piece of evidence for attesting the correctness of outsourced data. Providers and tenants form the blockchain network. Using such a technique gives two advantages. First, it adds trust, transparency, and security to existing integrity monitoring techniques and second users are directly participating in the process of securing their outsourced information system.

Given an SLA guaranteeing the integrity of users data, there are two main procedures, namely the *setup* and *verification* phases.

- *The setup phase* is performed before uploading a file to the cloud. In this phase, a tenant prepares *tags* for the data to be used later in the verification phase. *Tags* are proofs containing the hash of the data (see Section 6.2). The tags are used by the tenant to perform verification without the need to trust any party including the provider. After this process, the tenant sends the data and its hash value to a provider. The hash value is used by the service providers to check the correctness of the initial data upload. After successfully uploading the data, both the tenant and the provider publish the hash value of the data to the ledger.
- *The verification phase* is the procedure after the data is correctly uploaded and the hash value published in the ledger. A tenant sends a request for the provider to perform an integrity checking and receives the hash value computed over the current state of stored data. Using the tags generated in the previous phase a tenant can confirm the correctness of the stored data. If the data is not correct, a tenant can claim an SLO violation using the proof stored in the ledger and proceed with the next procedure as stated in the SLA.

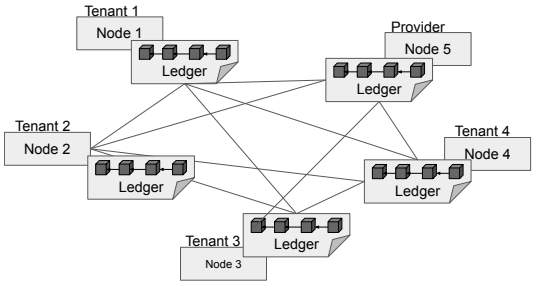


Figure 2: Consortium blockchain formed by tenants and

Section 6.2 presents a more detailed description of each phase. The blockchain is serving as a secure storage for hash values and the provider's infrastructure stores the actual data.

6.1 Architecture

There are five main components (see Figure 1): *an underlying blockchain, a smart contract (chaincode), a blockchain application, a service provider (cloud) and a user (tenant).*

- *The underlying blockchain* is the blockchain network which stores pieces of evidence (hash values) of data. The hash values are used as an anchor, referring to the original data. This component is assumed to be trusted and secure. The network is a consortium blockchain formed by the tenants and providers, i.e. a private blockchain having more than one authorized entity to add blocks (see Figure 2). Every participant in the network holds all valid blocks and updates the list according to the consensus algorithm. Our SLA verification method can be used with any type of consortium blockchain. The number of required participants in the network depends on the type of blockchain implementation used and the resiliency model requirement (e.g. using HLF with a crash fault tolerant consensus algorithm, to tolerate f number of failures requires $2f+1$ nodes). See Section 7 for implementation details.
- *A smart contract* is the logical and programmable component of blockchains to perform different tasks. It is the only component which directly interacts (performs read and write operations) with the underlying blockchain. Our smart contract consists of the following functions: *initLedger()* called only once to initialize the blockchain, *addData(data)* used to add '*data*' to the ledger, *queryData(data)* to check if '*data*' is

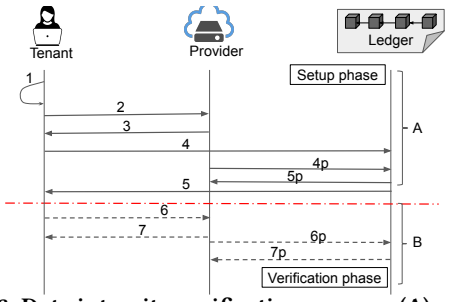


Figure 3: Data integrity verification process, (A) setup phase and (B) verification phase

in the ledger and *Invoke(f, param)* used by external applications to call a function from the smart contract.
To avoid duplicate entry when receiving a write operation from a tenant or provider, the smart contract first checks if the same data exists in the blockchain. If it found the same data it returns the block ID of the existing data. Otherwise it will add the requested data. Thus, even if both tenants and providers make a separate request using the same data, the data will be added only once, and both the tenant and the provider will hold the same block ID.

- *Blockchain application* is a module which acts on behalf of a user, i.e. the entity who wants to call functions from the smart contract. The caller can be either tenants or the provider to store or retrieve a piece of evidence for a data block. Our application contains the *storage module* used to store evidences in the ledger and the *verification module* to retrieve evidences. These modules call the *addData()* and *queryData()* functions respectively from the smart contract.
- *The service provider* is an entity providing the storage service. The provider offers SLAs to guarantee data integrity. As described in Section 5 providers are not malicious, and they can respond to requests (challenges) from a tenant. They can also perform verification in order to check the correctness of an SLO breach claim from tenants.
- *Users* are owners of the data stored in the cloud and they sign an agreement with the cloud provider. Users add evidences of data integrity into the ledger and they perform verification of data integrity to check if the SLO is still valid.

6.2 Integrity Checking Process

The process of checking integrity contains two phases, the *setup* and *verification* phases. In the setup phase, the tenant generates *tags* for later verification. In the verification phase, the actual checking is performed by using the evidence generated in the previous phase. Figure 3 shows the two phases; in addition we provide algorithms (Algorithms 1 and 2) describing both phases. The suffix 'p' in the figure represents the tasks performed by the provider.

6.2.1 Setup phase. Figure 3 (A) shows the setup phase and Algorithm 1 shows the procedure in the setup phase (the line numbers in the algorithm represent the arrow numbers in the figure).

1. The tenant (the data owner) performs an operation on the data to produce tags. In practice, this procedure is hashing the data using a hash algorithm e.g. SHA-x, or xxHash. Since we consider non-malicious inputs, a non-cryptographic hash algorithm is enough for our use case. By using this step the tenant generates the required tag values.
The tags include three parts: (i) the hash of the data, $H(D)$, (ii) n random strings called *nonce*, $(R_1, R_2 \ldots R_n)$, and (iii) the hash of the data concatenated with each random string, $(H(D + R_1),\ H(D+R_2)\ \ldots\ H(D+R_n))$. The number of random strings (*'n'*) is determined by the length of the SLA validity period and the frequency of verification. For example, if an SLA is valid for five years and verification is performed once per day, the tenant will generate 1825 (5 * 365) different strings and performs the hash of data plus a random string, for every value. The hashes and generated random values, i.e $(H(D + R_1),\ H(D + R_2)\ \ldots\ H(D + R_n))$ and $(R_1, R_2 \ldots R_n)$, should not be shared with other parties, especially with the provider. These nonce values are used to force providers into performing a fresh hash computation. Even if the method requires $(n + 1)$ hash operations, it is practical because of two major reasons: (i) the SLA has a validity period and (ii) data integrity is not a property which is checked very often, unlike availability. Thus, the variable 'n' is bounded.
2. The tenant uploads the data with the hash value, i.e. *(D, H(D))* to the cloud storage. Upon receiving the data, the service provider runs the same hash function over the data and compares the result with the provided hash value. If the value matches the provider sends a confirmation; otherwise, the provider assumes there is an error in the data transfer and notifies the tenant. In the case of such an error message, the tenant should repeat the process.
In practice this process is not novel. Amazon S3's [3] *'s3api put-object'* command takes *'--content-md5 and --metadata'* arguments and Amazon uses this information to perform integrity checking. Amazon confirms the correctness by returning an *'Etag'* and stores the hash value with the data. This step is done only once unless the data is changed. In that case, the hash of the new data should be computed.
3. Once the upload is confirmed the tenant publishes the hash value $H(D)$ in the ledger. Publishing a hash value can be achieved by using the storage module from the client application. Providers also publish $H(D)$ to the ledger. However, the smart contract prevents duplicate entries in the ledger; thus the second *addData()* operation returns the block ID of the previously added data. This way both the provider and the tenant have the same block ID.

6.2.2 Verification phase. At this stage, $H(D)$ is published, and both the tenant and provider have a block ID referring $H(D)$ in the ledger. Note that the strings $(R_1, R_2 \ldots R_n)$, and $H(D + R_1),\ H(D + R_2)\ \ldots\ H(D + R_n))$, are stored privately by tenants.

To perform verification, a tenant can challenge a provider to compute a hash value over the current state of the stored data. Figure 3 (B) and Algorithm 2 show the verification phase process.

6. A tenant selects a R_i and a file name to be checked and sends it to the provider. This value of R_i is then removed from the set of random numbers, i.e. it will be used only once.
7. The provider computes the hash of the data with the nonce (R_i) and returns the result to the tenant. The tenant compares

Algorithm 1: Setup phase

Input: Data 'D', Hash function '$H()$'
Result: n Random strings ($R_1, R_2 ... R_n$), Hash of 'D' $H(D)$ and Hash of 'D' with random strings ($H(D + R_1),\ H(D + R_2)\ ...\ H(D + R_n)$)

1 Generate n random strings, compute $H(D)$ and ($H(D + R_1),\ H(D + R_2)\ ...\ H(D + R_n)$) ;
2 Upload the data, i.e send $(D, H(D))$ to the service provider ;
3 The provider computes a fresh hash of 'D', $H(D)'$ and compares it with the received one;
if $H(D)' == H(D)$ **then**
  return 'success' ;
else
  return 'error' ;
end
4 **if** *the previous step is successful* **then**
  addData($H(D)$);
else
  return to step two and re-upload the data;
end
5 Get blockID, the blockID indicates where $H(D)$ is stored in the ledger

the return value with the locally stored value and concludes about the integrity of the data stored in the cloud.
In the event of a discrepancy between these two values, a tenant can claim for an SLO violation and use values from the ledger as evidence. Our integrity verification process guarantees that the tenant and the service provider will hold the same block ID value, i.e. they both refer to the same value in the ledger. Note that holding the same pointer to a secured data storage location does not automatically resolve a conflict between tenants and providers. However, it can help in the process to resolve a disagreement. One way of such a usage can be in the process of legal action.

Values written in the ledger are immutable, i.e. they are secured by duplication and the consensus algorithm; hence, the ledger is serving as a secure and trusted anchor for both tenants and providers.

Service providers can check the integrity of stored data, by hashing the data over its current state and comparing the result with the one stored in the ledger. This checking process is especially helpful when there is a complaint from tenants, and a provider wants to check the validity of such claims.

7 IMPLEMENTATION

We have a prototype implementation of the proposed data integrity SLA verification tool. This section presents the implementation details by describing each component listed in Section 6.1.

We used *Hyperledger Fabric* (HLF) [16] as a back-end blockchain. For our use case, the unique features of HLF provide three advantages over other implementations. HLF is permissioned, there is no native digital currency, and it allows writing a smart contract in any programming language. As a permissioned ledger, HLF allows us to create a consortium blockchain network with the tenants and the provider as the participants. The absence of native digital currency allows participants to perform operations on the ledger

Algorithm 2: Verification phase

Input: n Random strings ($R_1, R_2 ... R_n$) and Hash of 'D' with random strings ($H(D + R_1),\ H(D + R_2)\ ...\ H(D + R_n)$)
Result: *true* (no data corruption) or *false* (there is data corruption)

6 Select one R_i from the set of strings and send it to the provider. Remove R_i from the set of strings in order not to use it again ;
7 Compute $H(D + R_i)$ and return the *result* ;
if *result == Previously computed* $H(D + R_i)$ **then**
  return 'true' ;
else
  return 'false';
end

without transacting payments for each operation. Writing a smart contract using general purpose programming languages helps to develop smart contracts easily and rapidly. Additional benefits of HLF include its modular consensus, and its active community both from academia and industry. Note that other consortium blockchain networks can be easily adapted for our use case.

All the other modules are implemented using Python except the smart contract, which is written using the *Go* programing language. We used the *xxHash* [15] algorithm to perform hash operations. *xxHash* is an alternative to the SHA hash algorithm families. It is a non-cryptographic hash algorithm with better speed than SHA families. The main criterion for the hash algorithm is to avoid collisions and since we consider non-malicious inputs a non-cryptographic hash algorithm is enough for our use case. We use *xxhash.xxh64()* method in our implementation and incremental hashing based on a fixed block size is used rather than hashing the whole data at once to decrease the hashing time.

8 EVALUATION

This section presents the setup used to perform a performance evaluation of the proposed method and a discussion on the performance and the security of the proposed method.

8.1 Experimental Setup

We run our experiment on Grid5000 [10] testbed. Three physical nodes are used to represent a user, a provider and a consortium blockchain built using Hyperledger Fabric (HLF). Each physical node contains two Intel Xeon E5-2630 v3 CPU, eight cores per CPU and 128 GB memory, all running Ubuntu version 14.04.

The entire blockchain network runs on a single physical node with containerized services, i.e. participants and related components are instantiated using Docker containers [6], and they communicate through virtual networks. Our blockchain network contains ten participants. HLF uses a structure of organizations and peers, and in our network there are five participating organizations (i.e. each organization has two participants in the network). For our experiment, one node in the network is owned and managed by a tenant and one node by a provider. The remaining eight nodes can be seen as other tenants and providers participating in the network.

HLF is configured to use Kafka cluster and Apache ZooKeeper ensemble [16] implementing a crash fault tolerant consensus algorithm. There are four nodes in the Kafka cluster and three nodes

forming the ZooKeeper ensemble. In practice, since HLF is modular and our verification method is not dependent on the consensus process, any algorithm can be plugged and used, including Byzantine fault-tolerant algorithms. The network represents providers and tenants who agreed on SLA terms on guaranteeing the integrity of data outsourced to the cloud infrastructure.

A second separate physical node represents a Storage as a Service cloud provider. Users sign an SLA with the provider and submit their data after performing the setup process described in Section 6.2. In our experiment the provider service is based on python Simple HTTP server [13]. A third separate physical node is used to represent a user. Users perform the setup and verification process on this node.

The data size is a significant factor in our verification process because the time needed for operations like hashing is directly related to the data size. For our experiments, we used different data sizes ranging from 2GB to 16GB. This range of data sizes is enough to show our experiment goals, but in practice, cloud users can upload tens or hundreds of gigabytes of data to the cloud. Initially, all the files are not cached i.e. the cache is cold.

8.2 Performance Evaluation

The proposed SLA verification method requires two additional resources. First, time to perform the steps presented in Section 6.2. Second, at least one node to participate in the HLF blockchain network. The results of the performance evaluation are structured in three parts: the time required for operations in the setup phase, the overhead of the verification phase, and additional resources required to participate in the process.

In an environment where there is no integrity checking the only task is to upload the data to the cloud provider. Adding integrity checks requires two more tasks, namely hashing the data and publishing the hash value in the ledger. There are two kinds of hash operations performed by tenants: (i) hashing the data alone to be sent for the provider and be published in the ledger, and (ii) hashing the data with random nonce values. Figure 4 shows the time required for the first kind of hashing, publishing the hash value, and uploading data. Note that the time to publish a hash of the data is constant, since the output of a hash function have fixed size. Time to publish means the time required to write a hash value into the ledger. As shown in the figure it is relatively small (10% and 1% for 2 and 16 GB respectively) compared to the total time. However, our blockchain network is being simulated on a single physical machine. In a real production setup, the time could be higher since participants are physically distributed.

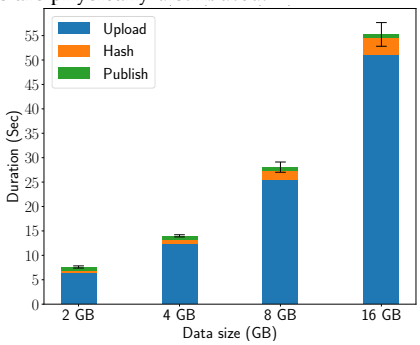


Figure 4: Time for upload, hash & publish tasks (setup phase)

The results of all our experiments are reported over an average of ten rounds. Writing to the ledger takes on average 0.725 seconds, almost twice the time required for a read operation. It is because writing to the ledger also performs a read operation to avoid multiple entries of the same data in the ledger.

The second type of hash consists in selecting n random strings and hashing the data with each of those strings. For example, with an SLA having a validity period of five years and a frequency of one verification per day, the tenant selects 1825 random strings and hashes the data with each string. In comparison, our method performs a smaller number of hash operations than other cryptographic solutions presented in Section 4.2. However, it takes more computation time than other solutions, because our method performs a hash of the whole data with each random strings while other solutions compute a hash of a few blocks out of the whole data. Parallelization can be used to optimize this process (see Section 9).

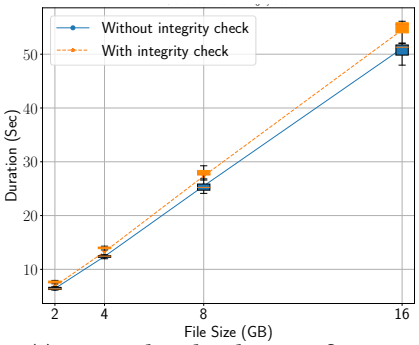
Without considering the second type of hash, Figure 4 shows that the time required for uploading data is dominant over other tasks. This task is not avoidable even without performing an integrity check. For the second type of hash, if a single hash operation takes t seconds, the total operation takes $(n \times t)$ seconds using a single process, where n is the number of random strings. For example, to hash a 2 GB and 16 GB files take 0.4564 and 3.4266 seconds respectively. However, this task is highly parallelizable and the time could be optimized to $\frac{(n \times t)}{p}$, where p is the number of processes used for the hash operations and it is bounded by the number of CPU cores available on the tenant's machine.

The verification phase consists of asking providers to perform a hash of the data with one random string. Hence, this hashing task is the only additional time compared to the baseline i.e. downloading the data without doing integrity checks. Figure 5 shows a comparison between the baseline (without integrity check) and checking integrity performed using the setup described above. The baseline operation (the solid blue line in Figure 5) measures only the time needed to download the given data while the integrity check (the dotted orange line in Figure 5) measures time to download the data plus the time for integrity checking operations. Doing integrity checks introduces an addition of around 6% of total time. The additional time is a result of hashing, and it is directly related to the data size which is also related to the baseline (download) time.

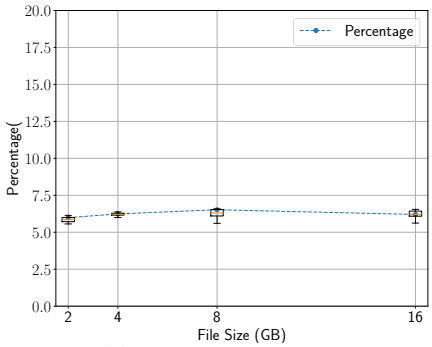
At least one node is required to participate in the HLF blockchain network. A tenant or a provider can join the HLF network as an organization with minimum requirements for operation. These include a node which participates in the ordering service (i.e. consensus process), a node acting as a peer to maintain the state and store a copy of the ledger, and a client which acts on behalf of the tenant and submit transactions. These can be achieved by using light containers provided by HLF, and they can be hosted in a machine having as low as 2GB memory. Regarding disk requirements to store the blockchain data, every entry in the ledger is a key-value pair (e.g 256 bytes for each key and value). This should be easily manageable by using regular personal computer storage devices.

9 CONCLUSION

The method proposed in this paper addresses the issue of openness in the SLA verification process. The method relies on a distributed ledger using consensus algorithm to keep an untampered state of



(a) Time with and without verification



(b) Overhead percentage

Figure 5: Time overhead as a result of integrity verification while downloading the data

the stored data. The process integrates security and trust by design. Tenants and providers can perform verification independently, and no one is dependent on another single entity. However other entities are required to keep the blockchain network running.

HLF is used in our prototype implementation with the default configuration i.e. without doing any optimization. If there is a need for high throughput, e.g. a large number of clients with frequent uploads, HLF can be optimized to handle more than 3500 transactions per second, according to Androulaki et al. [16]. Moreover, in our experiment, the blockchain network is entirely running on a single physical node using Docker containers and virtual networks. In practice, participants in a blockchain network are geographically distributed. Such distribution introduces additional latency, and our method should be further evaluated on this regard.

In our verification process, the given data is hashed with n random strings, i.e the process executes n hash operations. This verification process takes a longer time when compared to other integrity checking protocols. The hashing process can be optimized by using parallel processes, since the tasks of hashing the data with n different random strings are independent of one another.

In our threat model, we assume the existence of a secure communication between cloud providers, tenants, and ledger. In the absence of such secure communication system, a man-in-the-middle attack can affect our verification process and create a conflict between tenants and providers. For example, a malicious attacker can alter the data sent from the provider to the ledger. Such an attack can cause a conflict when a tenant claims an SLO violation.

The provider neither lies nor guesses the result in advance when asked to compute the hash of data with a given random value because the tenant uses a different nonce value for each verification. Service providers can also check the validity of SLO violation claims by computing the hash of the data over its current state and comparing the result with the one stored in the ledger.

We assumed that a tenant would keep some part of the generated proof private. If a tenant loses these values, it is impossible to proceed with the verification tasks. Hence, the tenant may require a highly available, secure, and private data storage mechanism.

On top of the proposed method, additional features can be added to the process. For instance, encryption can be used to add confidentiality. Blockchains can be further elevated to automate different tasks in the SLA life-cycle including payments for service and automatic compensation for SLA violation.

In this paper, we showed the advantage of having secure elements in the SLA verification process for data integrity. The secure element used in our case is a distributed ledger (blockchain). This secure component is highly programmable to perform different tasks; as a result, any remote data integrity checking protocol can be implemented following our method.

REFERENCES

[1] Affordable and easy to use decentralized cloud object storage. https://storj.io/
[2] Amazon Compute SLA. https://aws.amazon.com/compute/sla/
[3] Amazon S3. https://aws.amazon.com/s3/
[4] Chainpoint. https://chainpoint.org/
[5] Corrupt iCloud data causes iOS crash. https://www.macobserver.com/tmo/article/corrupt-icloud-data-can-cause-ios-springboard-home-screen-crash
[6] Docker. https://www.docker.com/
[7] DriveHQ SLA. https://www.drivehq.com/premium/DriveHQSLA.aspx
[8] Dropbox. https://www.dropbox.com/
[9] Facebook temporarily loses more than 10% of photos. http://bit.ly/2lwHkfU
[10] Grid5000. https://www.grid5000.fr/
[11] Guidance on GxP data integrity. https://www.gov.uk/government/publications/guidance-on-gxp-data-integrity
[12] On Public and Private Blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/
[13] Python HTTP servers. https://docs.python.org/3/library/http.server.html
[14] S3 data corruption. https://forums.aws.amazon.com/thread.jspa?start=0&threadID=22709&tstart=0
[15] xxHash. http://cyan4973.github.io/xxHash
[16] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proc. 13th EuroSys*. ACM.
[17] F. Apolinário, M. Pardal, and M. Correia. 2018. S-Audit: Efficient Data Integrity Verification for Cloud Storage. In *Proc. TrustCom/BigDataSE*.
[18] E. Gaetani, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone. 2017. Blockchain-based database to ensure data integrity in cloud computing environments. In *Proc. ITASEC*.
[19] Z. Hao and N. Yu. 2010. A multiple-replica remote data possession checking protocol with public verifiability. In *Proc. ISDPE*. IEEE.
[20] Z. Hao, S. Zhong, and N. Yu. 2011. A privacy-preserving remote data integrity checking protocol with data dynamics and public verifiability. *TKDE* 23, 9 (2011).
[21] Bin Liu, Xiao Liang Yu, Shiping Chen, Xiwei Xu, and Liming Zhu. 2017. Blockchain based data integrity service framework for IoT data. In *Proc. ICWS*.
[22] W. Luo and G. Bai. 2011. Ensuring the data integrity in cloud data storage. In *Proc. CCIS*. IEEE.
[23] S. Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf
[24] R. A. Popa, J. R Lorch, D. Molnar, H. J. Wang, and L. Zhuang. 2011. Enabling Security in Cloud Storage SLAs with CloudProof.. In *Proc. USENIX*.
[25] P. Wang, D. J. Dean, and X. Gu. 2015. Understanding real world data corruptions in cloud systems. In *Proc. IC2E*. IEEE.
[26] A. T. Wonjiga, L. Rilling, and C. Morin. 2018. Verification for security monitoring SLAs in IaaS clouds: The example of a network IDS. In *Proc. NOMS*.
[27] A. T. Wonjiga, L. Rilling, and C. Morin. 2019. *Defining Security Monitoring SLAs in IaaS Clouds: the Example of a Network IDS.* Technical Report RR-9263. Inria.
[28] Y. Zhu, H. Hu, G. Ahn, and M. Yu. 2012. Cooperative provable data possession for integrity verification in multicloud storage. *TPDS* 23, 12 (2012).